Axiomathes
https://doi.org/10.1007/s10516-021-09561-8

The Limits of Computation

Andrew Powell[1]

Received: 24 December 2020 / Accepted: 26 April 2021

Abstract
This article provides a survey of key papers that characterise computable functions, but also provides some novel insights as follows. It is argued that the power of algorithms is at least as strong as functions that can be proved to be totally computable in type-theoretic translations of subsystems of second-order Zermelo Fraenkel set theory. Moreover, it is claimed that typed systems of the lambda calculus give rise naturally to a functional interpretation of rich systems of types and to a hierarchy of ordinal recursive functionals of arbitrary type that can be reduced by substitution to natural number functions.

Keywords Computation · Lambda calculus · Type theory

1 Introduction

This paper concerns the sorts of functions that can be computed by a human or a finite machine. For the sake of definiteness, computable functions have to start with a finite input (coded as a natural number) and produce a finite output (a natural number again). This paper will concentrate on functions which always produce natural number output given natural number input (that is to say, the *total functions*), since they can be characterised in terms of logical systems more readily, and are desirable in practice (since no-one wants a program which given some input gets stuck in an endless loop and never returns any output).The emphasis is on the computable functions themselves in systems of the typed lambda calculus and corresponding logical systems. This paper does not include any material on higher order recursion theory, which is a very rich area (see Longley (2005) for a survey), primarily because of the emphasis of higher order recursion theory on relative rather than absolute computability, but also because higher order computations require a hypercomputer to realize (see Koepke and Seyfferth (2009) for a realization of ordinal recursion on an ordinal

✉ Andrew Powell
andrew.powell@imperial.ac.uk

[1] Institute for Security Science and Technology, Imperial College, South Kensington Campus, London SW7 2AZ, UK

Published online: 26 May 2021

computer with an infinite tape indexed by transfinite ordinals). That is not to say that computable total natural number functions do not involve higher order structures (such as impredicative logical systems and transfinite ordinals), but the emphasis is on what a finite computer (with unbounded memory) could actually compute. There is a fine line here, but the distinction lies at any point where the results of a previous computation that could not be performed by a finite computer (Turing machine) are required to complete a computation.

There is a focus in this paper on typed systems of lambda calculus. To informally introduce the typed lambda calculus, a *type* is a property of a set of objects. It is usual to allow that types and their objects are definable by finite expressions, the finite expressions defining objects being called *terms*. Terms in the lambda calculus are variables and (in some systems) constants, functions (written using the lambda symbol) and the *application* of functions to previously defined terms. The process of moving from the application of a function to a variable to the function is called *abstraction*, and is the inverse of application. In typed systems of lambda calculus, each term has a type. For example, $(\lambda x : A)(y : B)$ is a term representing the function that takes term (variable) x of type A to term (variable) y of type B. In some typed systems of lambda calculus types are only ever represented by variables, while in others constants are allowed, and in some systems expressions involving variables and constants are allowed. There is further explanation in this paper, but texts such as (Geuvers and Nederpelt 2014) and (Guallart 2015) provide a good introduction. For this paper another central concept is that of *reduction* of a lambda term. The idea is that a term is simplified as far as possible using function application (although there are often other rules, to deal with equality for example) and an irreducible *normal form* results. For example $(\lambda x : A)(y : B)(a : A)$ reduces to y for $y : B$ if y is a variable, while $(\lambda x : A)(\lambda y : B)(yx)(a : A)$ reduces to $(\lambda y : B)(ya)$ for $a : A$.

Typed systems of lambda calculus form a bridge between computer programs on the one hand and axiomatic deductive systems on the other. When lambda terms are reduced or in general converted to other terms, the process forms a computer program, while the rules for reduction or conversion correspond to logical rules (see Sect. 4 below). All the systems of typed lambda calculus described in this paper can give rise to functional programming languages.

The style of this paper is to discuss and build on key papers in the literature of computation theory and formal logic to show what has been achieved and then to make suggestions on how the computational systems described in these papers can be extended. These suggestions aim to characterise the sorts of formal systems and programming languages which could be developed to extend existing computational paradigms. The original content in this paper is contained in Sects 7 and 10, and comprises a translation of sets to types, the characterisation of a recursion schema that results by substitution in computable functions and the views on the limits of ordinal complexity that can be used to define computable functions. Sects 2–6 comprise a literature survey of what might be called *modern type theory*, while Sects 8 and 9 are critical reviews of homotopy type theory (an intensional theory of types) and the untyped lambda calculus respectively.

2 Gödel and Kleene: General Recursion and the Undecidability of Totality

It is a well known result due to Kleene (see Kleene (1943)) that computations in a formal language can be decided accurate or otherwise given the computation and the program instructions in a primitive recursive way by coding the input and output, the program and the computation as natural numbers. Kleene showed that any computable function that returns a value can be written as primitive recursive function that operates on a formally verified computation, usually written as $U((\mu y)T(e,x,y))$ for primitive recursive function U, input x, computation of least (μ) code y, program of code e and primitive recursive computation checking predicate T.

If we consider programs that always have a natural number output given natural number input, called *total programs*, we see that this fact can be expressed as $(\forall x)(\exists !y)(f(x) = y)$ for program f, where $(\exists !z)P(z) := (\exists z)(P(z) \wedge (\forall w \neq z)\neg P(w))$. We can clearly re-write this formula as $(\forall x)(\exists !y)(\exists !z)(z = U(y) \wedge T(e,x,y) \wedge (\forall w < y)\neg T(e,x,w))$, where e is a numerical program code for f, indicating that totality of a program has logical complexity $\prod_2^0$(*i.e.* can be written in the form $(\forall x)(\exists y)\ldots$, where the ellipses contain no further quantifiers), while the set of terminating programs and inputs has logical complexity $\sum_1^0$ (*i.e.* can be written in the form $(\exists x)\ldots$). Gödel showed in Gödel (1931) that no formal axiomatic proof system (that has quantifiers and can represent addition and multiplication of natural numbers), which is a type of computation system, can decide the truth of every proposition of quantifier complexity $\prod_1^0$ or richer (so $\prod_2^0$, or $\sum_2^0$ for example). Kleene showed that the characteristic function corresponding to $(\exists y)T(x,x,y)$ has undecidable totality (often called the "halting problem" and first shown undecidable for computers by Turing in Turing (1937)).

As important as these theorems are (and they are arguably the foundational results in mathematical logic and computability theory in the 20th Century), we must not overlook what Kleene (Kleene 1943) showed en route as far as computer programs are concerned. That is, deciding whether a computation is correct (via Kleene's T predicate) is in general computationally (far) easier in general than performing a computation (since computational correctness is decidable in primitive recursive arithmetic or weaker as it is expressible as a Δ_0^0 proposition, *i.e.* the proposition can be written with quantifier values *bounded* by the length of the expression, whereas the least y returned by $f(x)$ is an *unbounded* search operation. Ackermann in Ackermann (1928) showed that there are total programs which grow faster than any primitive recursive total program (as a nested double recursion)[1]; and it is possible to execute such a program, even though it will quickly exhaust the memory of actual computers. In fact Kleene's result shows that what makes a computable function hard to compute is the rate of growth of the function.

[1] See Tait (1961); Fairtlough and Wainer (1992) for the ordinal complexity of the Ackermann function and other nested recursions relative to primitive recursion. We can in fact see that n-nestd recustions for natural number n can be written as nested ordinal recursions up to ω^n and therefore as un-nested ordinal recursions up to ω^{ω^n}. Smorynski (1980) gives an overview of computable functions which grow much faster than the Ackermann function.

3 Gödel's System T and Girard's System F

The provably total functions (*i.e.* the programs which always return a result when given input and which can be proved to have that property in a formal axiomatic theory) of first-order Peano Arithmetic (including Ackermann's function that grows faster than any primitive recursive function) are precisely the hierarchy of functions reducible by application from primitive recursive functionals of finite type over the natural numbers (a result proved by Gödel in his famous *Dialectica* article Gödel (1958), Gödel (1994)) in a quantifier-free higher-order computation system called System T. To be more precise, Gödel showed that it is possible to compute a sequence of terms which witness any theorem of first-order Heyting arithmetic[2] (which applies equally to first-order Peano arithmetic by using Gödel's double negation translation or the direct method due to Shoenfield in Shoenfield (1967)), and since every non-empty term without free variables can be reduced by function application to a numeral and not to an empty term (a term in the empty type, the type of a contradiction),[3] first-order Peano arithmetic is consistent (see (Avigad and Feferman 1998) 4.2).[4] In the 1970s an extension of Gödel's System T of primitive recursive functionals of finite type led the way as a foundation for computing. Girard's System F (see Girard (1989)) is an extension of System T[5] which allows variables over and abstraction over types (known as *type polymorphism*). System F is a very powerful impredicative system[6] of the typed lambda calculus, having the same provable total natural number functions as second-order Peano arithmetic with a comprehension axiom for any formula of second-order arithmetic (see Girard (1989), Avigad and Feferman (1998)).

[2] First-order Heyting arithmetic is first-order Peano arithmetic with first-order intuitionistic logic.

[3] The reduction of any term without free variables to a numeral requires transfinite induction up to ϵ_0, see (Tait 1965b; Howard 1970), and for a generalisation to higher epsilon numbers for subsystems of second-order Peano arithmetic see (Howard 1980b).

[4] There is a game-theoretic version of the Dialectica interpretation in (Olivo 2008) which establishes the consistency of first-order Heyting arithmetic given System T.

[5] It is not clear cut whether System F is an extension of System T from Girard's Girard (1989) as it is described as the second-order type lambda calculus, but I follow Avigad and Feferman (1998) in taking System F to include all the axioms of System T.

[6] A formal system is *impredicative* if it is possible to define a predicate of the system in terms of a formula which includes the same predicate in the domain of the quantifiers of the formula. Impredicative systems can be inconsistent because Cantor style diagonal arguments are possible, but the remedy is to allow that the type of the diagonalized type is larger than the base type. Most impredicative definitions are mathematically fruitful, such as defining the least upper bound of a bounded set of real numbers defined as the intersection of all sets of real numbers which are upper bounds and fixed-points of monotone increasing operators. Poincaré, Weyl and Feferman (see Feferman (1988)) have argued impredicative definitions should be avoided, but they increase the power of a formal system greatly and do not lead to inconsistency unless you restrict the size of types. Of course, impredicative definitions do assume that the domain of quantification is well formed, and that the act of definition is actually picking out some particular object.

4 Curry-Howard Isomorphism

System F also explicitly pursues an idea due to Curry in Curry (1934) and Howard in Howard (1980a), that all computer programs can be regarded as a way of verifying the type of the data associated with a program. This idea is known as the *Curry-Howard isomorphism*. The same idea can be expressed in terms of proofs: a proof of a logical expression corresponds to a means of verifying the well-formed-ness of a type corresponding to the logical expression.

To be more specific, the logical rules enabling the formation, introduction and elimination of propositions and predicates in a natural deduction formulation of logic (see Gentzen (1969a), Prawitz (2006)) correspond to the rules enabling the formation, introduction and elimination of types, which in turn correspond to the rules for term rewriting in some typed version of lambda calculus. In Howard's Howard (1980a) he is able to show that a fragment of first-order predicate logic (actually intuitionistic Heyting Arithmetic in the second part of the paper) comprising $\wedge$, $\rightarrow$ and $\forall$ corresponds to pairing and projection and lambda abstraction and application introduction and elimination rules for implication for term (not type) variables (with it being possible to add other logical connectives).[7] It is also possible to represent types as propositions in first-order propositional logic (*i.e.* without quantification over propositions) for $\wedge$ and $\rightarrow$ but not for $\forall$. In System F Girard extends this correspondence at propositional level to introduction and elimination rules for universal and existential quantification for type variables by allowing lambda abstraction over types (written Λ),[8] which corresponds to "second-order" propositional logic with quantification over propositions. In fact since terms[9] can be coded as numbers, types are properties of numbers, and polymorphism allows the type of the natural numbers to be defined and quantified over (at least in the sense that "if the natural number structure exists, then ...).[10] System F is powerful enough to represent second-order

[7] To $r : P \rightarrow Q; p : P \Longrightarrow rp : Q$ corresponds $(P \rightarrow Q), P \vdash Q$; to $(x : P \Longrightarrow qx : Q) \Longrightarrow (\lambda x)qx : P \rightarrow Q$ corresponds if $P \vdash Q$ then $\vdash P \rightarrow Q$; to $p : P; q : Q \Longrightarrow \langle p, q\rangle : P \times Q$ corresponds $P, Q \vdash P \wedge Q$; to $\langle p, q\rangle : P \times Q \Longrightarrow p : P$ corresponds $P \wedge Q \vdash P$; to $qx : Q \Longrightarrow (\lambda x)qx : (\forall x)Q$, corresponds if $\vdash Q$ in the case where x is not free in any type symbol of a free variable of $q : (\forall x)Q$ then $\vdash (\forall x)Q$; and to $(\lambda x)qx : (\forall x)Q \Longrightarrow qx : Q$ corresponds if $\vdash (\forall x)Q$ then $\vdash Q$. For existential quantification, to $\langle t, qt : Q\rangle \Longrightarrow (\lambda x)qx : (\exists x)Q$ corresponds to $Q \vdash (\exists x)Q$ for t such that $qt : Qt$ is replaced by x in $(\lambda x)qx$; and to $(\lambda x)qx : (\exists x)Q \Longrightarrow \langle t, qt : Q\rangle$ for some term t, where term t replaces x in qx, corresponds $(\exists x)Q \vdash Qt$ in the case where t is not free in any type symbol of a free variable of $q : (\exists x)Q$. We can define $\vee$ by taking existential quantification over a finite domain (or taking the disjoint union of two types), and $\neg P := P \rightarrow \bot$, where $\bot$ is the empty type, *i.e.* the type with no terms. Howard himself in Howard (1980a) distinguishes between weak and strong existential quantification, weak existential quantification being equivalent to $\neg(\forall x)\neg P(x)$.

[8] We can define $A \wedge B := (\Pi C)((A \rightarrow B \rightarrow C) \rightarrow C)$, $A \vee B := (\Pi C)((A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C)$ and $\neg A := (\Pi C)(A \rightarrow C)$. Quantifier rules follow Howard in Howard (1980a) but taking $(\Lambda X)p : (\Pi X)P$ and $\langle U, u : P(U/X)\rangle : (\Sigma X)P$ and $(\Sigma X)P := (\Pi Y)((\Pi X)(P \rightarrow Y) \rightarrow Y)$, where Π is the universal type binder, Σ is the existential type binder and $P(U/X)$ means that U is substituted for free variable X in term P, see Girard (1989) Chapter 11.

[9] Terms of finite type of over the natural numbers can be written as finite sequences of functionals (with use of abstraction, function application and recursion).

[10] The natural numbers have type $(\Pi X)(X \rightarrow (X \rightarrow X) \rightarrow X)$ and $0 := (\Lambda X)(\lambda x : X)(\lambda y : X \rightarrow X)x$ and $s(t) := (\Lambda X)(\lambda x : X)(\lambda y : X \rightarrow X)y(tXxy)$, where tX is the application of ΛX to term t and Π is the type

Heyting Arithmetic as a deductive theory (with comprehension for arbitrary formulas in the language of second order arithmetic).

In Girard (1989) Girard showed System F to be a conservative extension for computable total functions of second-order Heyting Arithmetic (and therefore second-order Peano arithmetic). Girard's proof is very interesting because second-order Heyting Arithmetic (like System F) is an impredicative theory. Girard showed that a normal form always exists for all terms in System F and a reduction sequence that has a numeral as a normal form can be represented and proved to terminate in second-order Heyting Arithmetic, and conversely that all provably total functions in second-order Heyting Arithmetic correspond to reduction sequences in System F. Girard's argument quantifies over all sets of what he called reducibility candidates, which are reductions that avoid lambda abstraction, pairing and non-logical type construction (see Girard (1989), Pistone (2018)), the main argument being by induction on formula complexity, the comprehension axiom schema being used to show that sets of reduction candidates can be parameterized.

5 Martin-Löf's Intuitionistic Type Theory

Martin-Löf went further than Girard in Martin-Löf (1984) by allowing that types could be defined for any formula expressed in terms of logical connectives over individuals but not types (allowing that a type can mix types and terms, which is known as a *dependent type*). An example is that $(\forall x : A)(Px \rightarrow Qx)$ is a type, often written $(\Pi x : A)(Px \rightarrow Qx)$, where Π is the type binder corresponding to universal quantification. Similarly, $(\exists x : A)(Px \rightarrow Qx)$ is a type, often written $(\Sigma x : A)(Px \rightarrow Qx)$, where Σ is the type binder corresponding to existential quantification. It is the job of the mathematician according to Martin-Löf to show that a type (called a "proof" type) is non-empty using the standard (Heyting) interpretation of intuitionistic logic (see van Dalen (2001) for a description using the lambda calculus),[11] which establishes the validity of the inference represented by the type (under the Curry-Howard isomorphism). To be clear, according to this conception of type theory proofs inhabit types (represented by terms) and provide a validation of the logical inference represented by the type. This resulting theory, known as Intuitionistic Type Theory, is actually weaker in its mature form[12] than System F because type abstraction is

Footnote 10 (continued)
binder for universal quantification. System F can also define the Boolean type and the empty type. These and more complex types are defined in Girard (1989) Chapter 11. It can be seen that natural numbers in System F are explicitly symbolically constructed, so System F is not a purely logical system because it has the same domains of (first-and second-order) quantification as second-order Heyting Arithmetic.

[11] Proofs of existential propositions in intuitionistic logic must exhibit/construct at least one instance of the existential quantifier, while the rules for implication and universal quantification are methods for transforming a proof of the antecedent into a proof of the consequent. A proof of a conjunction of propositions is a pair of proofs of each conjunct, while the proof of a disjunction of propositions is a proof of either one of the disjuncts together with an indicator for which disjunct has been proved.

[12] Originally Martin-Löf had a universal type V with the property $V : V$, that is, that the type of V is V, which makes Intuitionistic Type Theory very powerful, although Girard showed in Girard (1972) that the resulting system was inconsistent.

not permitted. Intuitionistic Type Theory is a very clean foundation for intuitionistic mathematics in terms of computable functions because proofs are computations. Intuitionistic Type Theory includes inductively defined types (see Martin-Löf (1984) p. 42 *et seq*). An *inductively defined type* (or *inductive type*) is any type where terms of the type can be defined in respect to previous terms by a finitely definable relationship in a formal language, that is well-founded, *i.e.* every chain of definitions of terms is finite and returns a definite value.[13] Intuitionistic Type Theory with inductively defined types is a powerful theory (although the induction may need to terminate at the first non-recursively definable ordinal without the ability to form variables over types, in the sense of System F)[14].

Intuitionistic Type Theory also led to a view of the Curry-Howard isomorphism that is often called "Propositions as types",[15] that is to say that propositions correspond to types and proofs of propositions correspond to proofs of a type being non-empty by exhibiting a term that inhabits the type. The reason for this correspondence is that in Intuitionistic Type Theory every well formed first-order intuitionistic logical term has a type and *vice versa*, and closed terms correspond to propositions.[16]

Martin-Löf's Intuitionistic Type Theory is a beautiful foundational theory for computation and mathematics. The value of introducing dependent types should not be underestimated. With dependent types comes the ability to create *transfinite types* that System F[17] does not have. For example, if $[0] := N$ and $[n+1] := [n] \to N$, then $(\lambda n : N)[n]$ is a transfinite type (see Avigad and Feferman (1998) 10.1), and a function (term) $t : (\lambda n : N)[n]$ will be such that $t(n) : [n]$, *i.e.* the return type of the function is dependent on the value of natural number n. Dependent types also enable all logical connectives to be defined in a unified way.[18]

[13] Inductive definitions can be thought of as the least fixed point of a monotone operator on a set, see for example (Aczel 1977).

[14] See the discussion in the "Definable Types" section of this paper.

[15] There is a view that if a type is a proposition and a proposition has a type, then a type should have a type. The problem is that a type is not identical to a proposition, precisely because they have different properties such as a type applies to terms while a proposition does not apply to terms. The Curry-Howard isomorphism does of course establish that types under abstraction and application rules correspond to propositions making logical inferences.

[16] In Tait (2006) Tait argues by using the "propositions as types" correspondence in Intuitionistic Type Theory that Heyting's intuitionistic semantics is clear and not impredicative, and that intuitionistic semantics has a clearer foundation than System T. But in my view the axioms of System T are clear since they are designed to produce computations of terms witnessing the values of primitive recursive functionals, and they can be used in classical and intuitionistic systems.

[17] Transfinite types are not possible in Girard's more powerful System F_ω described in Section 6.

[18] As in System F we can define $A \wedge B := (\Pi C)((A \to B \to C) \to C)$, $A \vee B := (\Pi C)((A \to C) \to (B \to C) \to C)$ and $\neg A := (\Pi C)(A \to C)$, and for predicate logic $(\forall x : S)P(x) := (\Pi x : S)P(x)$ and $(\exists x : S)P(x)$ $(\Pi C)((\Pi x : S)(P(x) \to C) \to C)$, where $(\Pi x : S)P(x)$ is the dependent product type of types $P(x)$ for $x : S$, see Geuvers and Nederpelt (2014). As noted in Footnote 8 it is possible to perform quantification in System F, but quantification over types in System F is most naturally understood as propositional quantification via the Curry-Howard isomorphism.

6 Girard's System F_ω and Coquand's Calculus of Constructions

It is possible to extend System F to allow types to be defined with functional variables from any finite type to finite type, that is of the form $A \rightarrow B$, where A and B are built up from base types such as (generic type variable) $*$[19] (which is known as allowing *type constructors* over finite types), which gives the System F_ω. Type constructors can be regarded as defining a type variable by a formula involving type variables and type constants. In the same way that System F has the same provably total computable functions as second-order Peano Arithmetic, System F_ω has the same provably total computable functions as the union of all higher finite order deductive systems of Peano Arithmetic (see Girard (1973), Girard (1986)).

One of the most powerful type based system is Coquand's Calculus of Constructions in Coquand and Huet (1988),[20] which allows abstraction over terms, types and type constructors (see Geuvers and Nederpelt (2014) for a clear recent exposition). The Calculus of Constructions is a strict extension of System F and System F_ω because it allows types to depend on terms (allowing the introduction of transfinite types, see Sect. 5 above), as well as allowing abstraction over types and type constructors. The Calculus of Constructions is very elegant because of the symmetries between types and terms, in the sense that terms can be defined from terms (as in the simply typed lambda calculus)[21], terms can be defined from types (as in System F), types can be defined from terms (as in Intuitionistic Type Theory) and types can be defined from types (as in System F_ω). The Calculus of Constructions is at the top right of what is known as the Barendregt Cube (see Barendregt (1991) and Guallart (2015) for an overview).

There is a question though of whether there should be a correspondence between types and generic terms of those types (as the mapping between terms and a type is many-to-one). For example, if you allow a universal or generic type variable "Type" or "*", what is the generic term variable corresponding to the type "*"? The only sensible answer is that * is the type of a universal term variable "Term". But then you have to allow that Term is not also a type variable. In Coquand and Huet (1988) Coquand makes clear that the type of * is not *, but that does beg the question of what the type of * is. You could say that the smallest type of * is ** (or the "Type of Type"), but it is not difficult to see that you will end up with ramified types (admitting *, **, *** *etc.*), see Russell (1908), or with Russell's paradox (if you allow a type to be its own type). It is possible to say that the smallest type of * is something

[19] It is standard currently have a generic or universal type *, types such as $* \rightarrow *$ known as kinds by using type constructors, and a sort which is a universal kind in the formulation of System F_ω, see Geuvers and Nederpelt (2014).

[20] There is a further generalisation of the typed lambda calculus due independently to Berardi in Berardi (1988) and Terlouw in Terlouw (1989) called Pure Type Systems, but the generalisation does not always result in systems that have a consistent logic.

[21] The name comes from a typed lambda calculus that only allows abstraction over term variables and includes first-order predicate logic due to Church in Church (1950), but nowadays the name refers to a typed lambda calculus that includes a fragment of first-order propositional logic with the implication connective, see Geuvers and Nederpelt (2014) Chapter 2.

else, a sort say (often written ⋄). rather than a type **; but then the same regress occurs when you ask what is the sort of the sort ⋄. There is no suggestion that as a formal system that the Calculus of Constructions is inconsistent (after all "*" does not have to have a type), but for such a powerful type theory it is incomplete in terms of type assignment.[22] It is possible to create a hierarchy of types (often called a *type universe*) in the case where types are defined predicatively in terms of other types, and a hierarchy of universes emerge in order to refer to all of the types in the preceding universes.

It is tempting to say that "Type" and "Proposition" are large types (that is the type equivalent of a proper class), while individual types and propositions are small types. If that were the case then Proposition would not be a member of Type (or more naturally Proposition would not be a Type) because both would be the size of proper classes, which do not form a hierarchy of increasing cardinality (assuming the Axiom of Limitation of Size, see von Neumann (1925)). There are further strengthenings of the Calculus of Constructions such as the Calculus of Inductive Constructions Paulin-Mohring (2015), the Extended Calculus of Constructions Luo (1989) and the Extended Calculus of Constructions with inductive definitions Ore (1992), which add features to the universe of types to the Calculus of Construction (a countably infinite hierarchy of predicative types and a single impredicative layer of propositions that is lifted to the predicative type hierarchy).

7 Definable Types

In my view Intuitionistic Type Theory with impredicative type and functional variable (or type constructor) quantification and inductive type definitions would be a reasonable foundation of computation, which is equivalent to the Calculus of Constructions with inductive definitions.[23] One could allow any type for which an inductive definition could be given in terms of existing types, which can be used in further definitions of types. It is possible in Intuitionistic Type Theory to define well-ordered linear orderings which can be computed, the order types of which are represented by ordinals less than the first non-computable ordinal. The limit of complexity of computable linear orderings is the first non-computable ordinal, but tree structures (which are partial orders) can be much richer. Even a binary tree will define a type which for a tree of countably infinite height has $2^{\aleph_0}$ branches when each node has two successor nodes. Now $2^{\aleph_0}$ branches cannot be computed, but a complete binary tree with branches of height ω (often called *a universal fan*)[24] can be characterised uniquely as the smallest structure obtained by quantifying universally over all trees

[22] Of course set theory has exactly the same problem as type theory, which is why there is a "proper class of all sets" rather than "a set of all sets".

[23] Intuitionistic Type Theory and System F, both subsystems of the Calculus of Constructions, have inductive types, see Girard (1989) s. 11.5, p. 88 and Martin-Löf (1984), and Paulin-Mohring (2015) explicitly adds Inductive Types to the Calculus of Constructions.

[24] Strictly a universal fan allows for all choice sequences of natural numbers, but all natural numbers can be expressed as finite binary sequences.

that satisfy the inductive definition.[25] This could be achieved in System F or in an impredicative system which allowed quantification over dependent types (such as allowing type $(\Pi P)(\Pi x : X) \ldots P(x) \ldots$ to be formed from type P that depends on x, where Π is the type binder corresponding to $\forall$). In such a system very rich structures could be defined as types. For universal quantification over all structures satisfying a definition this means admitting lambda abstraction over types that represent the structures, which is a reasonable step to take. In practice the construction of a type does not need to specify the universal quantification explicitly, as this can be left implicit in the construction. For the purposes of computation this is sufficient, since a computer with any finite runtime would describe a universal fan of finite depth, which in the limit would give the universal fan. Admittedly the universal fan is not computable, but it is definable as a type.

The main reason why definable structures are relevant to computable functions is due to the relationship between formal deductive systems and computability. We know that structure definitions give rise to axioms that characterise those structures, and that any sufficiently rich formal deductive theory is incomplete with respect to deduction (see Sect. 2). Exactly the same is true of typed systems of lambda calculus. New structures (types) mean new terms to rewrite and to (try to) generate a normal form.

Tree based types can be used as a representative of *power types* (the type-theoretic version of the set of all subsets of a set, the power set, introduced in Cardelli (1988) as a way to capture the notion of generic *subtype*). By a process of iteration and abstraction at ordinal limits (producing a *limit type*), a type theoretic view of the cumulative hierarchy of set theory can be reproduced. If type identity is treated as extensional (that is, if two types have the same extension, namely $x : A$ if and only if $x : B$ for any term x of type A or of type B, then $A = B$),[26] and any sequence of term reductions is finite, then type theory is no different from set theory with the Axiom of Foundation[27], although types themselves and abstraction and application operators are more readily viewed as logics (via the Curry-Howard isomorphism) and functional construction processes than is the cumulative hierarchy of sets. The correspondence is presented in the table below. The axioms in the table are not a minimal set of axioms (as the Axiom schema of Replacement implies the Axiom schema of Separation), and the Axiom of Choice is not usually considered a core

[25] Here "all trees" is understood in the naive sense, but it is well known that second-order quantifiers can be interpreted as a fixed subset of the second-order domain of quantification, trees in this case if individuals are branches, see Väänänen (2001) for example for a discussion of Henkin semantics.

[26] Functions are not naturally extensional when considered as constructions as two constructions with the same extension are still distinct constructions, see Feferman (1985), and likewise for a type considered as a domain of functions. But it is harmless to consider functions with the same extensions on all domains as equivalent.

[27] In Krivine (2001) Krivine has shown how to extend the typed lambda calculus to Zermelo Fraenkel set theory (ZF) by translating formulas of ZF with an additional "strong membership" relationship into types, and Werner in Werner (1997) has shown how to interpret the Calculus of Constructions with inductive constructions in ZF set theory and *vice versa*.

axiom of first-order Zermelo Fraenkel set theory. The axioms cited however do correspond to distinctive types (Table1):

Type theory is also easier to use than set theory (see Farmer (2007)), although richer type theories (such as the Extended Calculus of Constructions and type theories with an intensional view of identity) can be extremely complicated. My own view is that mathematics is almost exclusively concerned with constructions carried out by functions, and type theory is correspondingly a natural foundation for mathematics and for computable functions in particular.[28] Type abstraction can be viewed as carrying the least amount of information greater than the information in any application of the type,[29] which is natural for types because abstraction requires the notion of a generic instance of a type[30] (for example a variable that comprises as a term of that type) but is not natural for sets viewed as collections.

8 Homotopy Type Theory

A recent development in type theory has been homotopy type theory (see Awodey (2012, 2013)). The basic premiss of homotopy type theory is that in the same way as there is a correspondence between propositions and types there is a correspondence between types and homotopy spaces. An homotopy space is a topological space equivalent up to homotopy (which roughly means that any two families of n-dimensional open neighbourhoods that cover the space can be deformed continuously into one another).[31] The promise of this approach is that new axioms of type theory will emerge that are fundamentally topological in nature. An example from Awodey (2013) is Whitehead's Principle (that a point-wise isomorphism between homotopy groups on well-behaved topological spaces is equivalent to the homotopic equivalence between the two spaces) applied to homotopy types. Homotopy type theory is also part of what is known as the univalence foundational programme for mathematics. Univalence seems to be the view that (homotopic) equivalence is what is meant by identity. That is to say, if identity is understood intensionally and identity of two terms needs a type, introduction and elimination rules and proof that terms are equal, then identity proofs of identity of terms can be viewed as paths in an homotopy space, identity between proofs of identity can be viewed as an homotopy between paths, and so on up the hierarchy of higher-order identities (of types, types of identity over types and so on) (see Awodey and Warren (2009)). This makes

[28] Of course category theory is also a function-theoretic foundation, but it is fundamentally algebraic rather than logical.

[29] An example is that for finite number n we can decide whether a binary sequence x of length n is a member of a set X of binary sequences of length n by means of a binary search with X in lexicographical order in n steps if the set is ordered lexicographically, whereas by adding another bit to the sequence the membership function applied to generic x can be decided in $n + 1$ bits.

[30] There is a link via the emphasis on terms rather than objects with the views in Parsons (1972), although Parsons does not explicitly consider the terms of systems of the lambda calculus.

[31] Strictly an homotopy type is a ∞-groupoid, which is a category in higher category theory that can be used to describe (among other things) homotopy between an ω-sequence of categories.

sense if identity of two terms is treated as needing a topological explanation that does not reduce to the syntactic equality of the two terms or to one term being a shorthand for another. But given that "propositions as homotopy spaces" is an analogy (or functor in category theory terms) in the same way "propositions as types" is, it is arguable that the term rewriting rules are as clear an explanation of identity of terms as it is possible to get.

It is difficult to judge how successful homotopy type theory will be, although there is no reason in type theory why a type should not be a geometrical or topological space, and it is unclear what the advantage is of treating every type as though it were a homotopy space (other than having a means to handle intensional identity of terms and types). There is no doubt a desire to put to use some very rich mathematical structures described in category theory which do not seem to be reducible to sets and have an inherent geometrical structure, but it is not yet known whether homotopy type theory introduces any new classes of computable function over and above those introduced by axiomatizations of existing areas of mathematics.

9 The Untyped Lambda Calculus and Domains

An alternative to systems of typed lambda calculus is to move away from the requirement of provable totality of functions and embrace programs which do not always terminate. In the untyped lambda calculus (due to Church) terms do not reduce to a minimal normal form in finitely many steps. This is attractive in light of the undecidability of the halting problem because the untyped lambda calculus is equivalent to a universal Turing machine in its power. Introducing a partial ordering where there is a tree of partial functions ordered by inclusion and having programs which always terminate as their limit is a powerful idea (due to Scott in Scott (1976), Scott (1993)). In fact Scott produced a model of the untyped lambda calculus (as a term-rewriting system not as a logic), showing that there are partially ordered sets which have the same structures as the continuous functions from the partially ordered set into itself (see Turner (1990) for a clear account). The problem is that the untyped lambda calculus does not always result in a normal form when the lambda terms are rewritten using a reduction rule (see Geuvers and Nederpelt (2014)), and does not result in a consistent logic. It is possible to produce a lattice based logic based on domains (see Abramsky (1991), Scott (1993) for example), but the significance of the axioms (or rules) is unclear as the axioms reflect the lattice structure rather than a logical inference system.[32]

[32] Of course Boolean and Heyting algebras are based on lattices of the same names, and any logic should have a notion of "or" and of "and".

Table 1 The correspondence between sets and types

Set theoretic axiom	Type theoretic equivalent
Empty set	Empty type
Union set	Disjunctive type[a]
Power set	Power type/Subtype
Separated set	Subtype defined by properties
Replacement set	Functional and limit types[b]
Choice set	Choice type
Infinite set	Natural number type

[a]Intersection sets/conjunctive types can be defined using the Axiom of Separation and the empty set.

[b]Functional types are defined from types and parameters (types or terms), where types and terms correspond to sets in set theory.

10 What Might Type Theory Look Like?

As suggested in Sect. 7, type theory should include a type-theoretic version of Zermelo Fraenkel set theory, arguably with intuitionistic logic.[33] Intuitionistic logic has the great advantage over classical logic that when an existence claim is made there must be a concrete method for constructing an object. But at this point we are going to abandon intuitionistic logic. The reason is that we can define double negation elimination (*i.e.* the inference $\neg\neg A$ to A) as an elimination rule for negation, which is true of "truth" if not of "proof". Doing that introduces a symmetry to the status of the "there exists" and "for all" quantifiers, *i.e.* $(\exists x : B)P(x) := \neg(\forall x : B)\neg P(x)$ and $(\forall x : B)P(x) := \neg(\exists x : B)\neg P(x)$.[34] We can define dependent type $(\Sigma x : B)P(x) := [(\forall x : B)(P(x) \to \bot) \to \bot]$, and a term $t : (\Sigma x : B)P(x)$ would have the form $g[(\lambda x : B)(\lambda p : P(x))f(p)]$ for terms f and g of type $(\Pi x : B)(P(x) \to \bot)$ and $(\Pi x : B)(P(x) \to \bot) \to \bot$ respectively, but this is problematic because no term has type $\bot$. It is better to define $t : (\Sigma x : B)P(x)$ in a second order way as $(\lambda A)(g[(\lambda x : B)(\lambda p : P(x))f(p)])$, for type variable A, dependent type $P(x)$ and terms f and g of type $(\Pi x : B)(P(x) \to A)$ and $(\Pi x : B)(P(x) \to A) \to A$ respectively, which is the Russell-Prawitz-Girard approach (see Rathjen (2018)). While $\neg(\forall x : B)\neg P(x)$ is not the same as $(\exists x : B)P(x)$ it is equivalent via double negation elimination.[35] It is also well known that classical negation can be interpreted in programming terms as a context continuation, *i.e.* capturing state information and then passing that as a parameter to a program (see Griffin (1989)).

[33] There are two well known axiomatic formulations of constructive or intuitionistic logic, Constructive Zermelo Fraenkel set theory, see Aczel (1978); Myhill (1973).

[34] Geuvers and Nederpelt in Geuvers and Nederpelt (2014) use intuitionistic negation translated into the Calculus of Constructions and add the law of the excluded middle and double negation elimination as (alternative) axioms.

[35] That is $\neg(\exists x : B)P(x) \to (\forall x : B)\neg P(x)$ by noting that $P(t) \to \bot$ for any witness term t (the details can be worked through using a Beth tableau for example, and then applying contraposition and double negation elimination, see Bell and Machover (1977) Chapter 9).

For the purposes of computation, any operation that we introduce must be computable in the sense that given a natural number input after finitely many steps we arrive at a natural number output (see Introduction above). The question arises how to treat functionals of higher type than $N \to N$. The view taken here is a functional needs to be absolutely computable in the sense that with constant previously defined input values (which could also be functionals of lower type) the natural number output can be computed in finitely many steps by means of a recursion in a well-founded structure indexed by ordinal numbers. In order to be computable a functional is replaced by a term in the typed lambda calculus and the ordinal recursion must have the form that the ordinal of the reduct of a term is strictly less than the ordinal of the term. In general form we admit the following recursion schema for ordinal recursive functionals (commas not being used and brackets only used to keep the text readable):[36]

$Fgf0 := f$ and $Fgf\beta := g\beta(\lambda\gamma : \beta)Fgf\gamma$, where f has type T, g has type $\alpha \to (\alpha \to T) \to T$ and Fgf has type $\alpha \to T$ for any type T over the natural numbers, N, and α an infinite ordinal, and term $Fgf\beta$ is a function of finitely many terms $Fgf\gamma$ depending on the choice of $g\beta$ for $\beta > 0$ and of f for $\beta = 0$. γ, F, f and g may contain parameters, which are taken to be the same parameters for each absorbed into type T.

Box 1: Recursion schema for computable functionals

The idea of this recursion schema is to incorporate a computable finite-branching finite-depth tree for the value of F at ordinal β. Using a technique made explicit in Terlouw (1982), a well-ordering $\prec$ of order type α can be thought of as a well-ordering $\prec_*$ of order type 2^α (using ordinal exponentiation) understood as follows. If $\prec$ is a well ordering of the ordinal numbers $< \alpha$, a limit ordinal for simplicity, then ordinal codes (under a primitive recursive coding function $\lceil\ \rceil$ such as the ordinal sum) of finite (strictly descending) sequences of ordinals $\langle x_{i<n<\omega}\rangle$ such that $x_j \prec x_k$ if $j > k$ can be given a lexicographical ordering $\prec_*$ defined as follows. $\lceil\langle x_{i<n}\rangle\rceil \prec_* \lceil\langle y_{i<m}\rangle\rceil$ if $(\exists k < n)[(\forall l < k)(x_l = y_l) \wedge (x_k \prec y_k)]$ or $(\forall l < n)(x_l = y_l) \wedge n < m$, $|\langle x_{i<n}\rangle| =_* |\langle y_{i<m}\rangle|$ if $(\forall l < n)(x_l = y_l) \wedge n = m$ and $\lceil\langle y_{i<m}\rangle\rceil \prec_* \lceil\langle x_{i<n}\rangle\rceil$ otherwise. We can think of a computation tree of ordinal complexity 2^β as a tree where all $F(\gamma)$ terms of ordinal $\gamma < \beta$ are substituted into g, the well-foundedness of the computation of $F(\beta)$ following from $2^\prec$-induction ($\prec$-induction for each descending finite sequence of terms). We can replace the $\prec_*$ well ordering with a higher type and well ordering $\prec$ as follows, after Terlouw (1982) for ordinals less than ϵ_0 (the first ordinal α such that $\omega^\alpha = \alpha$ using ordinal exponentiation)[37]: $H(Fgf0\beta)(\gamma) = Fgf\gamma$ if $\gamma \prec_* \beta$, $H(Fgf0\beta)(\gamma) = g\gamma(\lambda\delta \prec_* \gamma)Fgf(\delta)$ if $\gamma = \beta$, $H(Fgf0\beta)(\gamma) = 0_T$ if $\beta \prec_* \gamma$, $H(Fgf\zeta\beta)(\gamma) = H(H(Fgfp(\gamma,\beta)\beta)p(\gamma,\beta)(\beta + 2^{p(\beta,\gamma)}))(\gamma)$ if $p(\gamma,\beta) \prec \zeta$, and $H(Fgf\zeta\beta)(\gamma) = 0_T$ if $\gamma \succeq p(\gamma,\beta)$, where $p(\gamma,\beta)$ is the least ν such that $\gamma < \beta + 2^{\nu+1}$.

[36] The usual definition is in terms of a primitive recursive total ordering of an ordinal notation, see Tait (1965a), but this will not be possible fo uncountable ordinals. Of course, uncountable ordinals are not computable, but a countably infinite set of ordinal notations can be used, which name countably many uncountable ordinals.

[37] Terlouw uses numerical encodings of the well orderings $\prec$ and $\prec_*$, which makes the notation more complicated, but means that limit ordinals do not need to be considered.

0_T is a representation of the natural number 0 in type T. $HFgf$ has a higher degree than Fgf as $degree(a \to b)$ is defined by $max(degree(a)+1, degree(b))$, and Fgf has type $\alpha \to T$ and $HFgf$ has type $\alpha \to 2^\alpha \to (\alpha \to T) \to \alpha \to T$, and so $degree(HFgf) = max(degree(\alpha \to T)+1, degree(2^\alpha)+1) \geq degree(Fgf)+1$. What the recursion H does is to start from $\beta = 0$ the definition of F by $\prec_*$ recursion and nests the recursion along $\prec$ in the case of a successor ordinal $p(\beta,\gamma)+1$, corresponding to the double application of H needed to achieve $2^{p(\beta,\gamma)+1} = 2^{p(\beta,\gamma)} + 2^{p(\beta,\gamma)}$ in the $\prec_*$ recursive definition. While this specific functional recursion only works for ordinals less than ϵ_0, it does show that wherever there is a finite computation tree defined by a partial ordering on a well ordering α, an in general nested recursion can be performed along α by substitutionof previously computed terms while at the same being performed by unnested recursion along the ordering of the finite computation sequences, which has ordinal 2^α (or ω^α) (see (Tait 1961; Fairtlough and Wainer 1992)).

In general a recursive functional, G say, may not result in a computable function since in principle for every limit ordinal $G(\beta)$ for $\beta \geq \omega$ has infinitely many predecessor terms, $G(\gamma)$ for $\gamma < \beta$. (This is not true of H because the function p is strictly decreasing for suitable ordinal input.) To make G computable (when reduced by substitution to a function $N \to N$) we require that $G(\beta)$ depends on finitely many predecessor terms. We could use an ordinal notation system where every ordinal can be written as a computable function of smaller ordinals and find a suitable "predecessor" function (as in the discussion above), or we could follow a hierarchy of total recursive functions. In the latter case the standard way to enforce the finite ordinal dependency is to treat any ordinal $\beta < \alpha$ as a term representing a monotonically increasing one-to-one computable function $N \to \alpha$ such that α is the least upper bound of $\beta(n)$ for $n \in N$ (called *a fundamental sequence)* such that $\beta(n) < \beta$ for all $\beta < \alpha$. Then for a limit ordinal δ we could have $G(\delta)(n) = G(\delta n)(n)$. In a similar vein, it is possible to treat β as a term which accepts natural number or higher type input (which can be reduced to a natural number type by substitution). By treating an ordinal as a term, we can see that nesting functionals corresponds to ordinal exponentiation and for hierarchies of total computable functions diagonalization corresponds to limit ordinals.[38] The study of these computable functions and functionals has a very rich and current literature, see for example Fairtlough and Wainer (1992) and Rathjen (2006), and it is worth noting that ordinals in this literature are countable ordinals described in a notation system, and that having notations for (countably many) uncountable ordinals extends the hierarchy of total computable natural number functions.

There is a natural upper bound to recursions given by the information content of a typed lambda term when represented as a quantificational

[38] An example of diagonalization at limit ordinals is as follows. If type S is defined as follows, $S(0) = \emptyset$, $S(1) := N \to N$, $S(n+1) := N \to S(n)$ for $n : N$ then $(\lambda n : N)Sn$ has ordinal ω. If $f : (\lambda n : N)(Sn)$ then the diagonal function $g(m,x) = 1 + fmmx$ for $x : S(m-1)$ and $m \geq 1$ cannot be $= feyx$ for some $e : N$, where $y : N$, because we can set $m := e$ and $y := e$ to get $feex = 1 + feex$, contradiction. (Function application is left associative so that $fmmx = (((fm)m)x)$.)

deductive theory. Information content is defined by transfinite recursion over the type of natural numbers, N, by $info(N) := \aleph_0$, $info(P \to Q) := info(P) + 1$, where $+$ is cardinal addition. For a dependent type $R(a)$ for some term a of type T, $Info(R(a)) = Card(T)$, $Info(\Pi x : T)R(x)) = Info(\Sigma x : T)R(x)) = Card(T) + 1$ and $Info(\Pi R : T)R) = Info(\Sigma R : T)R) = Card(T) + 1$, where $Card(T)$ is the cardinality of type T. The reason for this definition is that predicates can be thought of as Boolean functions, *i.e.* as functions with value True or False. Functions are many-to-one in general, and thus the information in a function cannot be less than the information in the domain of the function. In the case of an infinite type T, taking the least upper bound of ordinals that could be assigned to a term of type T in the domain of a Boolean function, yields $info(T \to \{True, False\}) = Card(T) + 1$.[39] The information content has the property that it forms a natural upper bound for transfinite induction determined by the cardinality of the type. The idea is that any finite sequence of functional applications which reduces the type of the term to N can be viewed as reducing the information content from $Card(T) + 1$ to $\aleph_0$ in finitely many steps. Any recursion on a term of type T that has finitely many steps from an initial value has only finitely many additional steps over $Card(T) + 1$ and can therefore be regarded as a transfinite recursion up to $Card(T) + 1$.

In general we would have to use infinitary rules for computing the truth of a quantified proposition, so to be computable the best one can hope for is to extend Gödel's Dialectica interpretation to evaluate the truth value of a quantified proposition as a definition of a recursive functional by a finite process of term reduction. This is possible if the induction axiom of first-order Peano arithmetic is replaced by a principle of transfinite induction up to a certain ordinal, $\alpha \leq Card(T) + 1$, and all quantifiers are treated as many-sorted first-order variables. Then the Dialectica interpretation will result in quantifier-free ordinal recursive functionals up to the same ordinal, α, in the form of the schema in Box 1, since at limit ordinals in applications of the principle of transfinite induction the finite dependency condition must apply to ensure a finite sequence of terms witnessing theorems in the theory.

This Dialectica-based (System T) approach does not address impredicative axioms such as comprehension axioms except insofar as ordinals can be defined impredicatively. This may be sufficient if every type is isomorphic to an ordinal, which requires the Axiom of Choice and reduces a type to a set; but in general we should use an impredicative dependent type system of the typed lambda calculus such as the Calculus of Constructions with inductive definitions to define recursive functionals of transfinite type in the form of the schema in Box 1. To be clear, what is being suggested is an impredicative, dependent type extension of Gödel's System T where the impredicativity is used to define structures (such as ordinals) and the dependent types enable the hierarchy of transfinite types to be extended to these impredicative structures.

The table below gives some upper bounds on definition by transfinite recursion in various systems of the typed lambda calculus, comparing (Table 2) the (in general

[39] An example is that in a countably infinite domain T, each term can be labelled with a countable ordinal. The least upper bound on the information in T is the first uncountable ordinal, identified with the cardinal $\aleph_1$.

Table 2 A comparison of the cardinal and ordinal strength of theories of the typed lambda calculus

Theory	Cardinal upper bound	Computable bound
Simply typed lambda calculus[a]	$\leq \aleph_0$	ω
System T[b]	$< \aleph_1$	ϵ_0
ITT[c]	$< \aleph_1$	$\geq \Gamma_0$
System *Fd*	$< \aleph_2$	Not known
System[d]	$\leq \aleph_\omega$	Not known
CoC[e]	$\leq$ Proper class[f]	Not known
Zermelo Fraenkel Set Theory[g]	Proper class	Not known

[a]We can use the Curry-Howard isomorphism to note that the simply typed lambda calculus is a fragment of intuitionistic propositional logic. and note that Gentzen's proof of the consistency of intuitionistic propositional logic uses mathematical induction to show that cuts (detours in a derivation that involve propositions not in the premisses of the derivation) can be removed (see Gentzen (1969a)).

[b]System T was constructed to prove the consistency of first-order Peano Arithmetic, which has proof-theoretic ordinal ϵ_0 as proved by Gentzen in Gentzen (1969b). The normalization process can also be assigned the ordinal ϵ_0, by assigning ordinals to terms in a reduction sequences, see Howard in Howard (1970).

[c]This is Intuitionistic Type Theory (ITT) with n many predicative universes for all natural number $n > 0$ without inductive types, see Rathjen (2018) Theorem 7.1. With inductive types the proof-theoretic strength of ITT is the same as Kripke-Platek set theory with n many recursively inaccessible ordinals for all natural number $n > 0$, see Rathjen (2018) Corollary 6.7.

[d]Girard proved that System F has the same provably total recursive functions as full second-order Peano arithmetic, see Girard (1989). There is a partial result in Hinman (1978) VIII Theorem 3.7 that Δ^1_2 sets of natural numbers are characterised by ordinal recursion up to $\aleph_1$, where it is assumed that an oracle could compute Δ^1_2 sets of natural numbers.

[e]The reason why the Calculus of Constructions (CoC) is stronger than System F_ω is that CoC has dependent types that allow transfinite types to be formed. In Werner (1997) it is shown that CoC with inductive definitions (the Calculus of Inductive Constructions) is inter-interpretable with Zermelo Fraenkel Set Theory with countably infinite many inaccessible cardinals.

[f]Rathjen has claimed in Rathjen (2017) that the Calculus of Constructions with inductive definitions is equivalent in proof-theoretic strength to Intuitionistic Zermelo Fraenkel Set Theory with double negation forms of the Power Set and Separation axiom schemas, which may be weaker than Intuitionistic Zermelo Fraenkel Set Theory.

[g]This is clearly not a system of the typed lambda calculus, but it is added only for comparison.

uncountable) cardinal bound with the countable ordinal bound (the proof-theoretic ordinal) of the computable functions (where known).[40]

It may be surprising to see uncountable cardinals (and therefore ordinals) in this table, but they are present because in impredicative, higher order systems of the typed lambda calculus, and in the corresponding deductive theories, such ordinals are definable as types. $2^{\aleph_0}$ was defined in Section 7 above, and $\aleph_1$ can be defined impredicatively as the intersection of well orderings $\leq 2^{\aleph_0}$ which include an uncountable well ordering. The computation of the value (or normal form) of a particular

[40] Countable ordinal notations may include labels for uncountable ordinals, but the notational system is collapsed back to a countable ordinal in order to be computable.

term involving types will always be a finite computation involving term rewriting, but there is no ordinal limit on the types of the terms involved. The reason why all the systems of the typed lambda calculus do not complete the set of Π^0_2 arithmetical truths (and therefore the set of arithmetically definable total functions) is simply that all the term rewriting or deduction systems can be coded as operations on numbers, and all richer and richer types add is the ability to frame larger and larger sets of total functions of the natural numbers. The set of computably functions is countably infinite because there are countably infinite many finite computer programs, but the set of computable functions cannot be computable, otherwise it would be possible to diagonalize out of the set to produce a computable function not in the set.

11 Conclusions

This paper has argued that computable functions can be identified with the functions that correspond to reduction sequences of terms at least as strong as the the Calculus of Constructions. What one ends up with ultimately (at least with extensional identity of types) is a version of Zermelo Fraenkel set theory with restricted transfinite recursion and a functional rather than collection based semantics operating on generic variables or constants of a given type (which could be an inductively defined type such as a power type or a limit type). It is possible to go further and look at the computable functions of subtheories of second-order Zermelo Fraenkel set theory (that is with a second-order replacement axiom), but it is necessary to treat the set theoretic universe as a determinate whole, as a proper class. New set formation principles do arise from the theory of proper classes (see for example Hellman (1989), Welch and Horsten (2016) and Roberts (2017)), but given that the literature on large cardinals and reflection principles is very large, it is sufficient here to note that computable functions are not bounded by any deductive theory or system of the typed lambda calculus, only by the requirement that they form a non-computable countably infinite set.

References

Abramsky S (1991) Domain theory in logical form. Ann Pure Appl Logic 51(1–77)
Ackermann W (1928) Zum hilbertschen aufbau der reellen zahlen. Math Ann 99:118–133

Aczel, P. (1977) An Introduction to inductive definitions. In: Barwise J (ed) 'Handbook of Mathematical Logic', North-Holland, chapter C.7, pp. 739–780
Aczel P (1978) The type theoretic interpretation of constructive set theory. In: MacIntyre A (ed) Logic colloquium '77. North-Holland, Amsterdam, pp 55–66
Avigad J, Feferman S (1998) Godel's functional (Dialectica) interpretation. In: Buss S (ed) Handbook of Proof Theory. Elsevier, pp 337–405
Awodey S (2012) Type theory and homotopy, in 'Epistemology versus ontology'. Springer 183–201
Awodey S (2013) Homotopy type theory: univalent foundations of mathematics. Princeton University, Institute for Advanced Study
Awodey S, Warren M (2009) Homotopy theoretic models of identity types. Math Proc Cambridge Philos Soc 146(1):145–144
Barendregt H (1991) Introduction to generalized type systems. J Funct Prog 1(2):125–154
Bell J, Machover M (1977) A course in mathematical logic, North-Holland
Berardi S (1988) Towards a mathematical analysis of the Coquand–Huet calculus of constructions and the other systems in Barendregt's cube, Technical Report CMU-CS-88-131, Universita di Torino
Cardelli L (1988) Structural subtyping and the notion of power type, in 'POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages', Association of Computing Machinery, pp. 70–79
Church A (1950) A formulation of the simple theory of types. J Symb Logic 5(2):56–68
Coquand T, Huet G (1988) The calculus of constructions. Inf Comput 76:95–120
Curry H (1934) Functionality in combinatory logic. Proc Nat Acad Sci USA 20(11):584–90
Fairtlough M, Wainer S (1992) Ordinal complexity of recursive definitions. Inf Comput 99:123–152
Farmer W (2007) The seven virtues of simple type theory. J Appl Logic 6:267–286
Feferman S (1985) Intensionality in mathematics. J Philosop Logic 14(1):41–55
Feferman S. (1988) Weyl vindicated: Das Kontinuum 70 years later, In C. Cellucci and G.Sambin, eds, 'Temi i prospettive della logica e della scienza contemporanee', 1: 59–93
Gentzen G. (1969a) Investigations into logical deduction. Trans. M E Szabo from "Untersuchungen über das logische Schließen. I". Mathematische Zeitschrift. 39 (2): 176–210. 1935 and "Untersuchungen über das logische Schließen. II". Mathematische Zeitschrift. 39 (3): 405–431. 1935., In M E Szabo, ed., 'The Collected Papers of Gerhard Gentzen', Vol. 55 of Studies in Logic and the Foundations of Mathematics, North-Holland, pp. 68–131
Gentzen G. (1969b), New version of the consistency proof for elementary number theory. Trans. M E Szabo from "Neue Fassung des Widerspruchsfreiheitsbeweises für die reine Zahlentheorie", Forschungen zur Logik und zur Grundlegung der Exakten Wissenschaften, 4: 19–44 , In M. Szabo, ed., 'Collected Papers of Gerhard Gentzen', Vol. 55 of Studies in Logic and the Foundations of Mathematics, North-Holland, pp. 252–286
Geuvers H , Nederpelt R. (2014) Type Theory and Formal Proof, Cambridge University Press
Girard, J.-Y. (1972) Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur, PhD thesis, Université de Paris VII
Girard, JY (1973) Quelques résultats sur les interpretations fonctionnelles, In A. R. D. Mathias and H. Rogers, eds, 'Cambridge Summer School in Mathematical Logic 1971', Vol. 337 of Lecture Notes in Mathematics, Springer, pp. 232–252
Girard J-Y (1986) The system F of variable types, fifteen years later. Theoret Comput Sci 45:159–192
Girard J-Y (1989) Proofs and types. Cambridge University Press
Gödel K (1931) Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. I. Monatshefte für Mathematik und Physik 38(1):173–198
Gödel, K. (1958) 'Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes', Dialectica pp. 280–287
Gödel K. (1994) On an extension of finitary methods which has not yet been used, In S Feferman, ed., 'Collected Works, vol. III', Oxford University Press, pp. 271–280
Griffin TG. (1989), A formulae-as-type notion of control, In 'POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages', pp. 47–58
Guallart N (2015) An overview of type theories. Axiomathes 25:61–77
Hellman G (1989) Mathematics without numbers. Towards a Modal-Structural Interpretation. Clarendon Press
Hinman PG. (1978) Recursion-theoretic hierachies, Perspectives in Logic, Cambridge University Press
Howard W. (1970) Assignment of ordinals to terms for primitive recursive functionals of finite type, In R. V. A. Kino, J. Myhill, ed., 'Intuitionism and Proof Theory: Proceedings of the Summer Conference at

Buffalo N.Y. 1968', Vol. 60 of Studies in Logic and the Foundations of Mathematics, North-Holland, pp. 443–458
Howard W. (1980a) The formulae-as-types notion of construction, In J Seldon , J Hindley, eds, 'To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism', Academic Press, pp. 479–490
Howard W (1980b) Ordinal analysis of terms of finite type. J Symb Logic 45(3):493–504
Kleene S (1943) Recursive predicates and quantifiers. Transact Am Math Soc 53(1):41–73
Koepke P, Seyfferth B (2009) Ordinal machines and admissible recursion theory. Ann Pure Appl Logic 160:310–318
Krivine JL. (2001) 'Typed lambda-calculus in classical Zermelo-Fraenkel set theory.', Arch Math Logic, 40(3): 189–205
Longley JR. (2005) Notions of computability at higher types I, In STR Cori, A Razborov and C Wood, eds, 'Logic Colloquium 2000', pp. 22–142
Luo Z. (1989) ECC, and extended calculus of constructions, In 'Proceedings of the Fourth Annual Symposium on Logic in computer science', pp. 385–395
Martin-Löf P. (1984) Intuitionistic type theory, Studies in Proof Theory, Bibliopolis
Myhill JR (1973) Some properties of Intuitionistic Zermelo-Fraenkel set theory, In 'Proceedings of the 1971 Cambridge Summer School in Mathematical Logic 1', Vol. 337 of Lecture Notes in Mathematics, pp. 206–23
Olivo P (2008) An analysis of Gödel's Dialectica Interpretation via Linear Logic. Dialiectica 62(2):269–290
Ore CE (1992)'The extended calculus of constructions (ECC) with inductive types', Information and Computation Volume 99, Issue 2, August 1992, Pages 231-264 99(2): 231–264
Parsons C (1972) A plea for substitutional quantification. J Philos 68(8):231–237
Paulin-Mohring C. (2015) Introduction to the calculus of inductive constructions, In BW Paleo , D Delahaye, eds, 'All about Proofs, Proofs for All', Vol. 55, College Publications
Pistone P (2018) Polymorphism and the obstinate circularity of second order logic: a victims' tale. Bull Symb Logic 24(1):1–52
Prawitz D (2006) Natural deduction: a proof-theoretical study, first published, 1965 edn. Dover
Rathjen M (2006) The art of ordinal analysis, In 'Proceedings of the International Congress of Mathematicians'. European Mathematical Society 45–69
Rathjen M. (2017) 'On relating type theories to (intuitionistic) set theories', Talk given at UC Berkeley 5 May 2017, available online
Rathjen M (2018) Proof theory of constructive systems: inductive types and univalence. In: Jäger G, Sieg W (eds) Feferman on foundations', number 13 in 'Outstanding Contributions to Logic'. Springer, NewYork, pp 385–419
Roberts S (2017) A strong reflection principle. The Rev Symbc Logic 10(4):651–662
Russell B (1908) Mathematical logic as based on the theory of types. Am J Math 30:222–262
Scott DS. (1976) Data types as lattices, Programming Research Group PRG-5, Oxford University Computing Laboratory
Scott DS (1993) A type-theoretical alternative to ISWIM CUCH, OWHY. Theor Comp Sci 121:411–440
Shoenfield J. (1967) Mathematical logic, Addison-Wesley Publishing Company
Smorynski C (1980) Some rapdily growing functions. Math Intell 2(149–154)
Tait W W (1961) Nested recursion. Math Ann 143:236–250
Tait WW (1965a) Functionals defined by transfinite recursion. J Symb Logic 30(2):155–174
Tait WW (1965b) Infinitely long terms of transfinite type. In: Crossley JN, Dummett MAE (eds) Formal systems and recursive functions. North-Holland, London, pp 176–185
Tait WW (2006) Gödel's interpretation of intuitionism. Philosophia Mathematica 14(2):208–228
Terlouw J (1982) On definition trees of ordinal recursive functionals: reduction of the recursion orders by means of type level raising. J Symb Logic 47(2):395–402
Terlouw J. (1989) Een nadere bewijstheoretische analyse van GSTTs, Technical report, University of Nijmegen
Turing A (1937) On Computable Numbers, with an Application to the Entscheidungsproblem. Proc London Math Soc Series 2(42):230–265
Turner R. (1990) Truth and modality for knowledge representation, Pitman Publishing
Väänänen J (2001) Second-order logic and the foundations of mathematics. Bull Symb Logic 7(4):504–520
van Dalen D (2001) Intuitionistic logic, in 'The Blackwell Guide to Philosophical Logic'. Blackwell 224–257

von Neumann J, (1967) An axiomatization of set theory; Translated by J. van Heijenoort from Eine Axiomatisierung der Mengenlehre, Journal of die Reine und Angewandte Mathematik, (1925) 154: 219–240. In: van Heijenoort J (ed) From Frege to Godel: A Source Book in Mathematical Logic. Harvard University Press, pp 290–301

Welch PD, Horsten L (2016) Reflecting on absolute infinity. J Philosop 113(2):89–111

Werner B. (1997) Sets in types, types in sets, In M Abadi and T Ito, eds, 'TACS 1997: Theoretical aspects of computer software', Vol. 1281 of Lecture Notes in Computer Science, Springer, pp. 530–546